Storing Many Pieces of Data - Vectors

Vectors

Generally speaking, computers are powerful because they can store many pieces of data and execute many statements very quickly. Previously, we learned the for loop that allows us to execute many statements using succinct notation. But, the power of the for loop is limited if we can't store more than a few pieces of data. With only individual variable declarations, in practice, we are limited to storing maybe 10 or 20 pieces of data. (Yes, the syntax allows for us to create a nearly boundless amount of variables, but to be able to complete a program quickly and type it succinctly, there's a much smaller upper bound.)

A vector is a built in data type in C++ that allows us to store a list of arbitrary size, only limited by the working memory allocated for C++ programs. (Most compilers have this set at some default and in programming competitions, in most cases, this limit is explicitly stated. Typically these limits range from 256 MB to 2 GB.) A vector can store multiple pieces of data all of the same type. If we wanted to store all the test scores in a class, we might create a vector like this:

```
vector<int> scores;
```

A Quick Note on Includes

In order to use a vector, we would add the following include to the top of our program:

```
#include <vector>
```

In fact, to use each tool that is typical in competitive programming, there is often a separate include. In regular programming, it's good to explicitly state each different include file. But, since competitive programming values shorter programs and the time it takes to type them, a common practice of C++ competitive programmers is to use the following include:

```
#include <bits/stdc++.h>
```

This one include includes all of the tools that most competitive programmers use. So, when you learn C++ programming for other purposes, make sure to look up which include is necessary for each tool, but for competitive programming, it'll be easy enough to always do the following:

```
using namespace std;
#include <bits/stdc++.h>

int main() {
    // Program here.
    return 0;
}
```

Using a Vector

The first thing we'll usually want to do with a vector is add some values to it. The standard method/function that allows us to add an item to the end of a vector is `push_back(item)`,

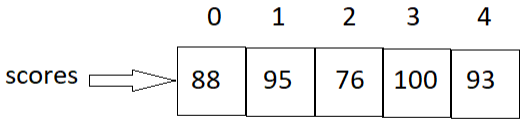
where item is the type of the items in the vector. (This type is specified in the <> when the vector is declared.)

A common way to specify input in a competitive programming problem is for the first number in the input, ***n***, to represent the number of values that follow, and then the ***n*** values appear, either one per line or space separated on the same or next line. Since cin naturally tokenizes, it doesn't really matter for our purposes which of these three possible formats is used. If we need to store all of these values, then we can do the following:

```
int n, tmp;
cin >> n;
vector<int> scores;

for (int i=0; i<n; i++) {
    cin >> tmp;
    scores.push_back(tmp);
}
```

Once we have some items stored in a vector, we can refer to those items using the [] operator, which indexes the vector. Here is a representation of what a vector with 5 scores looks like:



Since there are multiple variables stored in a single vector, we can't just use the expression scores to retrieve the value of a single score. Rather, we must also state which **index** in the vector scores we are trying to retrieve. We can think of a vector as a long street with addresses (called indexes) that start at 0 and end at the size of the vector minus 1. If we wanted to print out the second value stored in the scores vector, we can do:

```
cout << scores[1] << endl;
```

If we wanted to add all of the values in the vector and store that sum in a variable, we could do:

```
int sum = 0;
for (int i=0; i<scores.size(); i++)
    sum += scores[i];
```

Notice that we need to have a method to obtain the current size of a vector, since it can change. The method that returns the size of a vector is `size()`.

The preferred way to go through vectors is to use iterators, instead of explicitly indexing into them. (Though, for beginners, it's probably easier to visualize the indexing strategy shown above.)

The equivalent code to the code above that uses the iterator methods `begin()` and `end()` is:

```
int sum = 0;
for (auto x=scores.begin(); x!=scores.end(); x++)
    sum += (*x);
cout << sum << endl;
```

Technically, what scores.begin() does is return a pointer to the beginning of the vector. To get the value stored at the memory location where a pointer is pointing to, we use the * operator. (This complication is why perhaps the other method is easier for beginners to follow. But, this style is used so often, it's probably best to be introduced to it fairly early. Also, reading others' code is important, and many other people use this style of loop in C++.)

Ultimately, though, vectors aren't very useful if we can't modify values stored in them. We can simply do this using the assignment operator. For example, if we wanted to add 5 to the score of each student in the class, we could do:

```
for (int i=0; i<scores.size(); i++)
    scores[i] = scores[i] + 5;
```

Note that statements of the form

```
<var> = <var> <op> <expr>
```

Where var is a variable, op is an operator and expr is an expression of the same type as the variable are so common that C++ has the following shorthand (which also appears in many other programming languages):

```
<var> <op>= <expr>
```

Thus, the typical way in which the code above would be written is:

```
for (int i=0; i<scores.size(); i++)
    scores[i] += 5;
```

The following problem doesn't require a vector, but we'll store the values in a vector anyway, to illustrate a simple use of a vector and the necessary syntax.

Vector Example: Stand on Zanzibar

Here's a link to the problem:

https://open.kattis.com/problems/zanzibar

Basically, if we look between years, and there are a turtles one year and b turtles the next, if b > 2a, then b – 2a turtles must have been imported. Thus, once we read the data into a vector, we can loop through the vector, looking at turtles[i+1] and turtles[i], adding turtles[i+1]-2*turtles[i], if necessary. Here is the code:

```
using namespace std;
#include <bits/stdc++.h>

int main() {

    int nC, tmp;
    cin >> nC;

    for (int loop=0; loop<nC; loop++) {
        vector<int> turtles;
        cin >> tmp;
        while (tmp != 0) {
            turtles.push_back(tmp);
            cin >> tmp;
        }

        int res=0;
        for (int i=0; i<turtles.size()-1; i++) {
            if (turtles[i+1] > 2*turtles[i])
                res += (turtles[i+1]-2*turtles[i]);
        }
        cout << res << endl;
    }
    return 0;
}
```

Vector Functions Example

It is common to write functions that take in a vector and return some value based on that vector. There's a Kattis problem that is simply to fill in several functions with vectors:

https://open.kattis.com/problems/vectorfunctions

In this problem, the user has to fill in five functions, each of which takes in a vector:

```
// Reverse a vector.
// Note that it is sent as a reference, so you should
// reverse the same vector that was sent in.
void backwards(vector<int>& vec){
        ...
}

// Return every other element of the vector, starting with the
first.
// You should return a new vector with the answer.
// You are not allowed to modify the vector, even though it is
// sent as a reference. Therefore, the parameter is declared
"const".
```

```
vector<int> everyOther(const vector<int>& vec){
        ...
}

// Return the smallest value of a vector.
int smallest(const vector<int>& vec){
        ...
}

// Return the sum of the elements in the vector.
int sum(const vector<int>& vec){
  ...
}

// Return the number of odd integers, that are also on an
// odd index (with the first index being 0).
int veryOdd(const vector<int>& vec){
  ...
}
```

While there are "automatic" built in ways to do these things, when learning, it's best to look at solutions to these functions from the ground up. Besides, each of these is pretty short.

Let's take a look at the first function. The goal is to reverse the contents of a vector. Notice that if we swap pairs of elements, first and last, second and second to last, and so on, until we get halfway through the list, and then our list will be reversed. Here is an illustration for a list with 7 elements:

3	2	8	9	4	1	7	(swap index 0 and 6 →)
7	2	8	9	4	1	3	(swap index 1 and 5 →)
7	1	8	9	4	2	3	(swap index 2 and 4 →)
7	1	4	9	8	2	3	done…no point in swapping index 3 and 3...

To swap two variables, x and y, we need three lines of code:

```
int tmp = x;
x = y;
y = tmp;
```

Note: There are a few clever ways to do this with two lines of code, but these are more difficult to read. What is shown above is the standard way in which a swap is taught and should be coded for readability.

Now, we are ready to look at the solution to the backwards function:

```
void backwards(std::vector<int>& vec){

    int n = vec.size();
    for (int i=0; i<n/2; i++) {
        int tmp = vec[i];
        vec[i] = vec[n-1-i];
        vec[n-1-i] = tmp;
    }
}
```

Note that in this code, we'll have to refer to vec.size() several times. Since this is longer to type than n, in competitive programming, it's helpful to store this quantity in a variable with a shorter name. Oddly enough, this makes it less likely you'll make errors, especially with that quantity has to be part of an array index. Hopefully you can see that

`vec[i] = vec[n-1-i];` is easier to read than `vec[i] = vec[vec.size()-1-i];`

Thus, we must go halfway through the vector, and swap the appropriate pairs of elements. We just have to do a bit of index math (n – 1 – i) to find our "mirror" item to swap with.

One thing to note is that in the framework provided for the problem, the syntax lists

```
std::vector<int>& vec
```

instead of

```
vector<int>& vec
```

The ampersand ensures that a pointer is passed to the vector, so that changes made to the vector in the function are reflected in the vector that was passed in as a parameter. Secondly, we typically place the line:

```
using namespace std;
```

at the top of our programs so that we don't have to type "std::" before every item that would otherwise require it. The Kattis framework did not assume this was typed, which is why each function written will have this prefix every time a vector is declared.

The next three functions are fairly straightforward (easier than this first one), so the solutions are simply presented below without detailed explanation:

```
std::vector<int> everyOther(const std::vector<int>& vec){
    std::vector<int> res;
    for (int i=0; i<vec.size(); i+=2)
        res.push_back(vec[i]);
    return res;
}
```

```
int smallest(const std::vector<int>& vec){
    int res = vec[0];
    for (int i=0; i<vec.size(); i++)
        if (vec[i] < res)
            res = vec[i];
    return res;
}

int sum(const std::vector<int>& vec){
    int res = 0;
    for (int i=0; i<vec.size(); i++)
        res += vec[i];
    return res;
}
```

Notice that the first one requires some basic index math and to copy items into a newly created vector, which then gets returned. The next two follow a standard framework for iterating through vectors, solving two common problems. Note the mechanisms used in each as it's very common to have to either add a bunch of numbers of find the minimum or maximum of a bunch of numbers.

The last function is slightly more difficult. It asks us to count the number of indexes where both the index is odd and the value stored at that index is odd. Thus, similar to the second function, we want to go through every other array item, but instead of starting at index 0, we want to start at index 1. For each of these items, we want to ask if it's odd, which is equivalent to checking if the remainder when it's divided by 2 is 1. Here's the solution:

```
int veryOdd(const std::vector<int>& suchVector){
    int res = 0;
    for (int i=1; i<suchVector.size(); i+=2)
        if (suchVector[i]%2 == 1)
            res++;
    return res;
}
```

While much of what will be done in programming competitions with vectors will be more difficult than this, it's important to first get familiar and comfortable with vectors via standard operations.

Idea of a Used Vector

Often times it's useful to keep track of whether or not a particular item has been seen in a list. If the values we are looking for are integers in a relatively small range, then we can create either a vector of boolean or int, where we set index i to false if the value i isn't in the list and true if it has.

We can use this idea to solve the following problem:

https://open.kattis.com/problems/knotknowledge

In this problem, two input lists are given. The first list has distinct integers all in between 1 and 1000. The second list has all of the numbers from the first list, except one. The problem is to identify the missing number in the second list.

We can set up a vector of size 1001 (so indexes upto 1000 are valid) and set each value in the vector to 0.

Then, whenever we read in a number from the original list, we can change that index from 0 to 1.

When we read in the second list, we can change all of the 1s back to 0s.

Finally, the index that still has a one represents the missing number.

Here is this solution (note that many methods to solve this problem exist) coded up:

```
using namespace std;
#include <bits/stdc++.h>

int main() {

    int n, tmp;
    vector<int> seen;

    for (int i=0; i<=1000; i++)
        seen.push_back(0);

    cin >> n;

    for (int i=0; i<n; i++) {
        cin >> tmp;
        seen[tmp]++;
    }

    for (int i=0; i<n-1; i++) {
        cin >> tmp;
        seen[tmp]--;
    }

    int res = -1;
    for (int i=0; i<=1000; i++)
        if (seen[i] == 1)
            res = i;

    cout << res << endl;
    return 0;
}
```

Idea of a Frequency Vector

Hinted in the previous solution is the idea that if we have a list of many numbers, but instead just need to keep track of how many times each value occurred in the list, we can simply store, in index i, the number of times the value i appeared in the list. For many problems, storing data in this manner is preferable to storing the original list of numbers, particularly when the original list of numbers are all within a small range and contain many duplicates.

For example, if the original data was:

2, 3, 4, 4, 3, 1, 2, 4, 0, 2, 4, 4, 5

then the corresponding frequency vector would be as follows:

1	1	3	2	5	1
0	1	2	3	4	5

More Advanced Vector Problem: Slide Count

We'll include one more problem that uses a vector in these notes that's more difficult than the previous ideas. The problem is stated here:

https://open.kattis.com/problems/slidecount

The algorithm is given to us, but we just need to implement it.

We can first read in the original values into a vector.

As we simulate the algorithm, we need to keep track of two things for each number:

1) when it gets added to a range
2) when it gets deleted to a range

The difference between the second value and the first is the number of ranges that value was a part of. In the first sample we have the list

1 1 1 2 2

with a target value of 3. Let's say our time, t starts at 0. Then, we have that

index 1 gets added at time 0
index 2 gets added at time 1
index 3 gets added at time 2
index 1 gets deleted at time 3 (because adding the 2 in index 4 makes the sum too big)
index 2 gets deleted at time 4
index 4 gets added at time 5
index 3 gets deleted at time 6
index 4 gets deleted at time 7

index 5 gets added at time 8, and
index 5 gets deleted at time 9

Notice that if we subtract, for each index, the time it got added from the time it got deleted, we produce the correct output of:

```
3
3
4
2
1
```

Thus, we can create two vectors: one to store the time at which an index was added, and one to store the time at which an index was deleted. One key note about vectors that wasn't previously included: if we know how big we want our vector to be in the beginning, we can make it that size without adding new elements to it at all, like so:

```
vector<int> myvector(n);
```

where we assume that n is an integer variable storing a value representing how big we want myvector to be.

Here is the corresponding solution. Comments have been left in for explanatory purposes.

```
// Arup Guha
// 5/31/2023
// Solution to problem: Slide Count
using namespace std;
#include <bits/stdc++.h>

int main() {

    int n, target, tmp;
    cin >> n >> target;

    vector<int> nums;

    // Read in the numbers.
    for (int i=0; i<n; i++) {
        cin >> tmp;
        nums.push_back(tmp);
    }

    // Vectors to store start and end.
    vector<int> sTime(n);
    vector<int> eTime(n);
```

```
    int s = 0, e = 0, total = 0;

    // Weird set up but we can do this.
    for (int t=0;s<n; t++) {

        // This is the condition in which the end pointer moves.
        if (e<n && total+nums[e] <= target) {

            // Store the time.
            sTime[e] = t;

            // Update our running total.
            total += nums[e];

            // end boundary moves.
            e++;
        }

        // Here we move the start boundary.
        else {

            // This is an end time.
            eTime[s] = t;

            // Now our total becomes smaller.
            total -= nums[s];

            // Move up.
            s++;
        }
    }

    // Output result.
    for (int i=0; i<n; i++)
        cout << eTime[i] - sTime[i] << endl;

    return 0;
}
```